```
Proposal for C23
WG14 2918

Title:                Queryable pointer alignment
Author, affiliation: Alex Gilding, Perforce
Date:                 2022-01-25
Proposal category:   New features
Target audience:     Library developers
```

## Abstract

A pointer has an alignment that it satisfies, and an object type has an alignment that it requires. But C does not provide a portable way to check whether a pointer satisfies the alignment requirement for an object type. This means it is not currently safe to convert pointers (usually `void *`) to other types without making at last one unverified assumption.

If C added a Standard Library function that tested whether a pointer satisfied an alignment, it would be possible to portably know if a pointer is safe to convert.

# Queryable pointer alignment

```
Reply-to:           Alex Gilding (agilding@perforce.com)
Document No:        N2918
Revises Document No: N2852
Date:               2022-01-25
```

## Summary of Changes

### N2918

- Provide full alternative signatures; move declaration from `stddef.h`; add discussion of void-pointer alignment; null alignment; typo fixes; C23 draft version update

### N2852

- original proposal

## Introduction

C provides a concept of object *alignment*: "an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated".

However, while object types may impose an alignment requirement, there is no *portable* way to query the alignment of a generic address to see whether it can safely (in the absence of other concerns) be converted to a pointer to an object with the aligned type.

We propose that a standard library function is added to abstract the implementation-defined mapping between pointers and integers so that programmers can check pointers for alignment without needing to know about the addressing structure of the platform target.

## Rationale

Some types, most commonly floating types and atomic types, have strict alignment requirements. Others may have no hard alignment requirement but provide better performance when placed on their natural boundaries.

Unfortunately when a pointer that is not already strongly typed as pointing to an object of an aligned type is received, there is no way to check whether it is suitably aligned to convert to a pointer to a type with a stricter alignment, without knowing details of the target platform's addressing structure - the exact details that C's pointer types, pointer arithmetic, and mapping between integer and pointers, are supposed to abstract and protect the user from.

There is no obvious way to query this either without resorting to runtime derivations that would severely impact the performance of a conversion. Since the mapping is fixed to the entire platform adding any kind of runtime element to it is extremely wasteful.

However, the authors of a Standard Library for any given target must know what the addressing structure for that target is. It is possible to express this query correctly in native C code as long as the implementation details of the target are known, so the Library should provide a query that uses this secret platform knowledge so that users are not forced to arbitrarily guard entire program sections behind a "this probably only works on x86 *shrug*" with a low-performance fallback path.

Many users assume that a) pointer conversion to integer is a no-op, and that b) all addresses proceed in a linear "big array" indexed fashion. This is true for the most popular platforms like x64, but is not true on many embedded targets or less common platforms. For instance, the Cray T90 uses a word-addressing structure and provides C-compatible byte addressing by conveying the byte offset within a 64-bit word in the *topmost* three bits of the pointer. A no-op conversion from the pointer representation to a 64-bit integer will provide surprising results to an x64-user as it increments the byte address.

An x86-compatible alignment query (loosely, "what is the lowest set bit?") would not provide even remotely correct results on such a platform. The user has no way to determine what conversion and what operation on the integer representation is correct from within the portable C code; they can at most rely on identifying the platform and hoping they remembered that the address structure is different.

## Proposal

The Standard Library should provide a single function, tentatively named `memalignment`, which accepts any pointer as an argument and returns a `size_t` containing the maximum possible alignment supported by the given address. This acts as a dynamic counterpart to the static, type based `_Alignof` operator.

The result can be compared to an alignment requirement for a type obtained from `_Alignof`; if it is greater or equal, the pointer is suitably aligned to point to an object of the desired type.

For example:

```
// Assumes adding atomic qualification is acceptable
void want_atomic (void const * p) {
  size_t atomic_int_align = _Alignof (_Atomic int);
  size_t p_align = memalignment (p);
  if (p_align >= atomic_int_align) {
    _Atomic int * ai = (_Atomic int *)p;
    ...work with ai...
  }
}
```

A possible implementation of `memalignment` that would work on x64 might be:

```
size_t memalignment (void const * p) {
  intptr_t ip = (intptr_t)p;
  return ip & -ip; // convert to LSB value
}
```

This returns zero when the pointer is null, because it cannot be usefully used to access aligned memory (the user would need to perform a null test either way, so this is more defensive).

Because x64 does provide a flat address structure, the alignment, which is always a pure power of two per 6.2.8 paragraph 4, is described by the least significant value bit. Other architectures would provide other implementations as appropriate.

The question was asked whether a pointer to `void` preserves the same alignment requirements as the typed pointer it was converted from. We do not consider any additional clarification needed for the purposes of this library feature – a pointer to `void` is required (C23 6.2.5 “Types” p30) to have the same representation as a character pointer, i.e. it is required to be a byte-address. This doesn’t preclude the converted-from pointer from being word-addressed and the conversion to `void *` from being a representation-changing operation; but it does imply that the abstract address itself is maintained. The resulting `void *` must still compare equal to the pointer it converted from and compare unequal to a pointer to any other byte in the same or any other object.

Alignment being defined in terms of the byte distance between pointers, this means there is already enough normative wording to establish that the alignment of a pointer is stored in the converted `void *` representation. It is merely stored in an inaccessible, secret-platform-knowledge way.

## Alternatives

An alternative might be to provide a function that accepts a pointer and an alignment and returns a true/false value describing whether the pointer is sufficiently aligned for the alignment. This might provide a more concise macro interface. This would also hide considerations of over- aligned pointers from the user.

Another alternative might be to provide a function-like macro that accepts a type as the target alignment to compare against rather than an alignment value.

For example:

```
#define isaligned(P, T) (memalignment (P) >= _Alignof (T))
```

Which option is most user-friendly is largely subjective. The basic `memalignment` function provides the most information to the user, but the uses for that information beyond turning it into a true/false - especially if the result is beyond the implementation extended-alignments - are probably very limited.

The names `memalignment` and `isaligned` are suggested because they are already in the reserved namespace for memory/string functions. Names beginning with `alig*` are not currently reserved. We do not wish to drive the identifier-reservation explosion unnecessarily.

The proposal places the new function in `stddef.h`. Availability in freestanding is important. Should the proposed `stdbit.h` header gain acceptance, it might also make sense to be placed there.

Previously we proposed returning the maximum possible alignment when the argument is a null pointer. Although a pointer to any type can be null, regardless of the alignment requirement , the result didn’t make much sense because it implies that the address can be used to access aligned memory. Returning zero adds a layer of defensiveness  by indicating that the pointer cannot access *objects* of the required type, and is consistent with the null check the user requires anyway if they are working with nullable-pointers.

It also removes a branch, which users tend to care about quite a lot.

## Prior art

Prior art for this functionality exists and is mostly aimed at C++.

C++ proposal n4201 provided `is_aligned`, which was not adopted.

Boost.Align provides `is_aligned`.

Microsoft provide `IS_ALIGNED`.

All three of these correspond to the alternative proposal above `isaligned`, accepting two arguments and returning true if the first, a pointer, is aligned enough to satisfy the second, an alignment.

## Impact

The impact on compiler authors is zero because the feature can be implemented entirely in target aware C code. Converting a pointer to an integer and inspecting it is implementation-defined so the Library authors should know what the correct algorithm to use is. The maintenance impact is zero because this will not change once established for any given target.

## Proposed wording

Changes are proposed against the wording in C23 draft n2731.

Two alternatives are proposed, to standardize either `memalignment`, `isaligned`, or potentially even both.

Alternative 2 is not really necessary if Alternative 1 is adopted (it is trivial to implement in userspace as a wrapper), but Alternative 1 requires an external symbol to be added to the library because it is a function definition.

### Alternative 1

Add a new section to 7.22 "General utilities `<stdlib.h>`":

> **7.22.9** Alignment of memory
>
> **7.22.9.1** The `memalignment` function
>
> **Synopsis**
>
> ```
> #include <stdlib.h>
> size_t memalignment(const void* p);
> ```
>
> **Description**
>
> The `memalignment` function accepts a pointer to any object and returns the maximum alignment it satisfies. The alignment may be an extended alignment and may also be beyond the range supported by the implementation for explicit use by `_Alignas`. If so, it will satisfy all alignments usable by the implementation. The value returned can be compared to the result of `_Alignof`, and if it is greater or equal, the alignment requirement for the type operand is satisfied.

**Returns**

The alignment of the pointer `p`, which is a power of two. If `p` is a null pointer, an alignment of zero is returned.

**NOTE** an alignment of zero indicates that the tested pointer cannot be used to access an object of any type.

## Alternative 2

Modify 7.15 “Alignment `<stdalign.h>`”, paragraph one:

The header `<stdalign.h>` defines ~~four~~ **five** macros.

Add a new section to 7.15 “Alignment `<stdalign.h>`”:

**7.15.1** Alignment of memory

**7.15.1.1** The `isaligned` macro

**Synopsis**

```
#include <stdalign.h>
_Bool isaligned(pointer, type);
```

**Description**

The `isaligned` macro accepts a pointer to any possibly-qualified object and a type specifier for an object type, and returns a value indicating whether the pointer is an address sufficiently aligned to point to an object of the specified type. The parameter *`type`* shall be any type name suitable for use as the operand to the `_Alignof` operator.

**NOTE** A pointer which is over-aligned for the specified type satisfies its alignment requirement.

**Returns**

If `pointer` is not a null pointer and satisfies the alignment requirement for `type`, `isaligned` returns `true`. Otherwise, `isaligned` returns `false`.

**NOTE** a null pointer cannot be used to access an object of any type, and therefore does not satisfy any alignment requirement.

**EXAMPLE 1** the `isaligned` macro accepts an object type to point to as its second argument, rather than the type of the pointer to align.

```
#include <stdalign.h>

void f1 (void * p) {
  _Bool aligned = isaligned (p, int);
  if (aligned) {
    int * ip = p; // isaligned returned true for int,
    ...           // so p can convert to int *
  }
}
```

**EXAMPLE 2** the isaligned macro may accept a pointer to a different type than the specified object type.

```
#include <stdalign.h>

void f2 (long * lp) {
  if (isaligned (lp, float)) { // different object type
    float * fp = (float *)lp;  // have the same alignment
    ...                        // do something with that fact
  }
}

void f3 (int * ip) {
  if (isaligned (ip, _Atomic int)) { // different qualification
    _Atomic int * aip = (_Atomic int *)ip;
    ... // do something implementation-defined with that fact
  }
}
```

## References

- C23 n2731
- Cray T90
- C++ n4201
- Boost.Align
- IS_ALIGNED